

SOLID: o que é e quais os 5 princípios da Programação Orientada a Objetos (POO)

Fique por dentro do que são os princípios SOLID e como podemos aplicá-los nos códigos para aumentar a qualidade dos projetos.

A qualidade de software é um fator fundamental para o sucesso no desenvolvimento de qualquer projeto.

Afinal, aplicações bem projetadas, com boas práticas, códigos limpos e organizados, não só reduzem a incidência de erros, como também facilitam a manutenção e expansão ao longo do tempo.

E aí surge uma questão importante: *o que podemos fazer para aumentar a qualidade do software?*

Não tem uma única resposta para essa pergunta. Mas, dentre tantas possibilidades, uma opção é que seu código siga os princípios SOLID.

Neste artigo, vou explicar quais são esses princípios e como podemos aplicá-los para aumentar a qualidade dos projetos. Além disso, vou explorar o Clean Code e as boas práticas de programação.

Vamos lá?

O que é SOLID?

O acrônimo SOLID representa os cinco princípios que facilitam o processo de desenvolvimento — o que facilita a manutenção e a expansão do software.

Estes princípios são fundamentais na programação orientada a objetos e podem ser aplicados em qualquer linguagem que adote este paradigma.

Os 5 princípios são:

- S — Single Responsibility Principle (Princípio da responsabilidade única)
- O — Open-Closed Principle (Princípio Aberto-Fechado)
- L — Liskov Substitution Principle (Princípio da substituição de Liskov)
- I — Interface Segregation Principle (Princípio da Segregação da Interface)
- D — Dependency Inversion Principle (Princípio da inversão da dependência)



Origem dos princípios SOLID

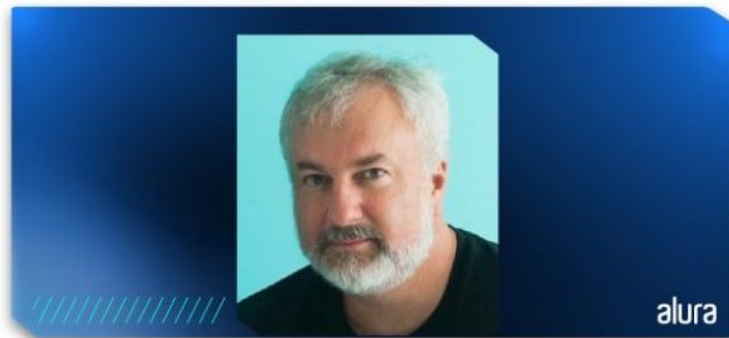
O primeiro indício dos princípios SOLID apareceu em 1995, no [artigo](#) “*The principles of OoD*” de Robert C Martin, também conhecido como “Uncle Bob”.

Nos anos seguintes, Robert se dedicou a escrever mais sobre o tema, consolidando esses princípios de forma categórica.

E, em 2002, lançou o livro “*Agile Software Development, Principles, Patterns, and Practices*” que reúne diversos artigos sobre o tema.



No entanto, a sigla SOLID só foi apresentada mais tarde, por Michael Feathers.



Agora que você já sabe o significado da sigla e a origem dos princípios, é importante dar um passo para trás para compreender o paradigma da Programação Orientada a Objetos (POO). Afinal de contas, como você já sabe, através da POO é possível aplicar os princípios SOLID.

O que é Programação Orientada a Objetos (POO)?

A [Programação Orientada a Objetos \(POO\)](#) é um paradigma que se baseia no conceito de “objetos”, que representam instâncias concretas ou abstrações de conceitos do mundo real.

Vamos aplicar essa ideia em um exemplo para compreender melhor. Olha só essa imagem, que representa um sistema de identificação de carros:

Perceba que a classe “carro” representa um modelo abstrato do que um carro precisa ter: o tipo, cor, placa e número de portas.

A partir desse modelo (que é a classe), podemos criar representações mais específicas de carros, que são os objetos:

- Tipo: Porsche
- Cor: Branco
- Placa: MHZ-4345
- Número de portas: 4

ou

- Tipo: Ferrari
- Cor: Vermelho
- Placa: JKL-0001
- Número de portas: 4

Assim, se for necessário incluir mais informações sobre os carros — como, por exemplo, se tem ar condicionado ou não, podemos mudar a classe.

A partir disso, todos os objetos serão obrigatoriamente atualizados com um novo campo que deve ser preenchido.

Nesse sentido, o paradigma promove a reutilização de código e facilita a manutenção desde sistemas mais simples, tal qual o exemplo, até os mais complexos.

Vale dizer que a Programação Orientada a Objetos é utilizada em muitas linguagens de programação populares, como [Java](#), Python, PHP, C++, C#, entre outras.

Agora que você já está por dentro do conceito de POO, está na hora de aprender sobre cada um dos princípios SOLID.

➤ Princípio da Responsabilidade Única (S - Single Responsibility Principle)

Para entender o princípio da responsabilidade única, vamos pensar no desenvolvimento de um gerenciador de tarefas. Vamos começar com o seguinte código:

```
public class GerenciadorTarefas {  
    public String conectarAPI(){  
        //...  
    }  
    public void criarTarefa(){  
        //...  
    }  
    public void atualizarTarefa(){  
        //...  
    }  
    public void removerTarefa(){  
        //...  
    }  
    public void enviarNotificacao(){  
        //...  
    }  
    public void produzirRelatorio(){  
        //...  
    }  
    public void enviarRelatorio(){  
        //...  
    }  
}
```

Problemática

Tente enumerar todas as funções que a classe GerenciadorTarefas tem. Ela é responsável por lidar com todas as operações das tarefas em si e também está consumindo uma API, enviando notificações para pessoas usuárias e ainda gerando relatórios da aplicação.

Pense na Orientação a Objetos. Um objeto gerenciador de tarefas deveria enviar e-mails e gerar relatórios? **Não!** Um gerenciador de tarefas **gerencia as tarefas**, não e-mails ou relatórios.



Como na imagem, não é legal que uma única ferramenta tenha 1001 utilidades, pois pode causar confusão.

Isso se intensifica quando falamos de desenvolvimento de softwares. O ideal é criar várias ferramentas diferentes, cada uma com sua própria função.

Solução

Para resolver esse problema vamos criar classes diferentes, cada uma representando uma função.

Nossa classe GerenciadorTarefas terá apenas o código relacionando a operação com tarefas. Outras operações estarão em outras classes. E cada classe será responsável por uma parte diferente da aplicação.

Assim, teremos a classe GerenciadorTarefas refatorada:

```
public class GerenciadorTarefas {  
  
    public void criarTarefa(){  
        //...  
    }  
    public void atualizarTarefa(){  
        //...  
    }  
    public void removerTarefa(){  
        //...  
    }  
}
```

Assim, vamos criar uma classe para consumir uma API externa, outra classe para enviar notificações e uma última classe para lidar com os relatórios.

```
public class ConectorAPI {  
    public String conectarAPI() {  
        //...  
    }  
}
```

```
public class Notificador {  
    public void enviarNotificacao() {  
        //...  
    }  
}
```

```
public class GeradorRelatorio {  
    public void produzirRelatorio(){  
        //...  
    }  
    public void enviarRelatorio(){  
        //...  
    }  
}
```

Talvez você se pergunte se as classes não são pequenas demais. Nesse caso, não estão. Cada classe reflete exatamente a responsabilidade que ela tem.

Se precisarmos adicionar algum método, por exemplo, relacionado ao consumo da API, vamos saber exatamente em qual parte do código devemos ir. Ou seja, fica muito mais fácil alterar o que for preciso.

Definição do Princípio da Responsabilidade Única

Em resumo, o princípio da responsabilidade única diz que: *“Cada classe deve ter um, e somente um, motivo para mudar.”*

Se uma classe tem várias responsabilidades, mudar um requisito do projeto pode trazer várias razões para modificar a classe. Por isso, as classes devem ter **responsabilidades únicas**.

Esse princípio pode ser estendido para os métodos que criamos também. Quanto mais tarefas um método executa, mais difícil é testá-lo e garantir que o programa está em ordem.

Uma dica para aplicar o princípio na prática é tentar nomear suas classes ou métodos com tudo que eles são capazes de fazer.

Se o nome está gigante, como GerenciadorTarefasEmailsRelatorios, temos um sinal de que o código pode ser refatorado.

Vantagens de aplicar o Princípio da Responsabilidade Única

Existem vários benefícios ao aplicar esse princípio, principalmente:

- Facilidade para fazer manutenções
- Reusabilidade das classes
- Facilidade para realizar testes
- Simplificação da legibilidade do código

➤ Princípio Aberto-Fechado (O - Open Closed Principle)

Para entender o Princípio Aberto-Fechado (a letra O da sigla), vamos pensar que estamos trabalhando no sistema de uma clínica médica.

Nessa clínica, existe uma classe que trata das solicitações de exames. Inicialmente, o único exame possível é o exame de sangue. Por isso, temos o código:

```
public class AprovaExame {
    public void aprovarSolicitacaoExame(Exame exame){
        if(verificaCondicoesExameSangue(exame))
            System.out.println("Exame aprovado!");
    }
    public boolean verificaCondicoesExameSangue(){
        //....
    }
}
```

Agora, precisamos incluir uma nova funcionalidade ao sistema: a clínica vai começar a fazer exames de Raio-X. Como incluir isso no nosso código?

Uma alternativa seria verificar qual o tipo de exame está sendo feito para poder aprová-lo:

```
public class AprovaExame {
    public void aprovarSolicitacaoExame(Exame exame){
        if(exame.tipo == SANGUE){
            if(verificaCondicoesExameSangue(exame))
                System.out.println("Exame sanguíneo aprovado!");
        } else if(exame.tipo == RAIOS) {
            if (verificaCondicoesRaioX(exame))
                System.out.println("Raio X aprovado!");
        }
    }
    private boolean verificaCondicoesExameSangue(){
        //....
    }
    private boolean verificaCondicoesRaioX(){
        //....
    }
}
```

Problemática

A princípio parece tudo certo, não é mesmo? Nosso código executa normalmente e conseguimos adicionar a funcionalidade corretamente.

Mas, e se além de raio-x, a clínica passasse a fazer também ultrassons? Seguindo a lógica, iríamos adicionar mais um if no código e mais um método para olhar condições específicas do exame.

Essa definitivamente não é uma boa estratégia. Cada vez que incluir uma função, a classe (e o projeto como um todo) vai ficar mais complexa.

Por isso, é necessário uma estratégia para adicionar mais recursos ao projeto, sem modificar e bagunçar a classe original.

Solução

Nesse cenário, o projeto compreende vários tipos de aprovação de exames. Assim, podemos criar uma classe ou uma interface que representa uma aprovação de forma genérica.

A cada tipo de exame fornecido pela clínica, é possível criar novos tipos de aprovação, mais específicos, que irão implementar a interface. Assim, podemos ter o código:

```
public interface AprovaExame{
    void aprovarSolicitacaoExame(Exame exame);
    boolean verificaCondicoesExame(Exame exame);
}
```

```
public class AprovaExameSangue implements AprovaExame{
    @Override
    public void aprovarSolicitacaoExame(Exame exame){
        if(verificaCondicoesExame(exame))
            System.out.println("Exame sanguíneo aprovado!");
    }
    @Override
    boolean verificaCondicoesExame(Exame exame){
        //....
    }
}
```

```
public class AprovaRaioX implements AprovaExame{
    @Override
    public void aprovarSolicitacaoExame(Exame exame){
        if(verificaCondicoesExame(exame))
            System.out.println("Raio-X aprovado!");
    }
    @Override
    boolean verificaCondicoesExame(Exame exame){
        //....
    }
}
```

Agora, como a interface representa a aprovação de um exame, para incluir mais um recurso ou mais um tipo de exame, basta criar uma nova classe que implementa a interface AprovaExame. Essa classe vai representar como o novo exame é aprovado.

Repare que sempre será possível implementar a interface AprovaExame ao adicionarmos recursos. Essa interface, no entanto, não muda. Estamos estendendo-a, mas não alterando.

Definição do Princípio Aberto-Fechado

Assim, é possível definir o Princípio Aberto-Fechado como: *“entidades de software (como classes e métodos) devem estar abertas para extensão, mas fechadas para modificação”*.

Ou seja, se uma classe está aberta para modificação, quanto mais recursos adicionarmos, mais complexa ela vai ficar.

O ideal é adaptar o código não para alterar a classe, mas para estendê-la. Em geral, isso é feito quando abstraímos um código para uma interface.

Aplicando o *Open-Closed*, é possível deixar o nosso código semelhante ao mundo real, praticando de maneira **sólida** a orientação a objetos.

Pense em um caminhão: toda a sua implementação, como motor, bateria e cabine é fechada para modificação.

Porém, podemos estender as tarefas que ele realiza dependendo da carroceria que anexamos, conforme mostra a figura abaixo:



Vantagens de aplicar o Princípio Aberto-Fechado

Ao aplicar esse princípio, é possível tornar o projeto muito mais flexível. Adicionar novas funcionalidades torna-se uma tarefa mais fácil.

Além disso, os códigos ficam mais simples de ler. Com isso tudo, o risco de introduzir bugs diminui de forma significativa.

Além disso, esse princípio nos faz caminhar diretamente para a aplicação de alguns padrões de projeto, como o [Strategy](#).

Assim, alinhamos várias boas práticas de desenvolvimento. O resultado disso é um código cada vez mais limpo e organizado.

➤ Princípio de Substituição de Liskov (L - Liskov Substitution Principle)

Para entender o Princípio de Substituição de Liskov (a letra L da sigla), vamos pensar no seguinte cenário: o desenvolvimento de um sistema de uma faculdade.

Dentro do sistema, há uma classe-mãe Estudante, que representa um estudante de graduação, e a filha dela, EstudantePosGraduacao, tendo o seguinte código:

```
public class Estudante {
    String nome;
    public Estudante(String nome) {
        this.nome = nome;
    }
    public void estudar() {
        System.out.println(nome + " está estudando.");
    }
}
```

```
public class EstudanteDePosGraduacao extends Estudante {
    @Override
    public void estudar() {
        System.out.println(nome + " está estudando e pesquisando.");
    }
}
```

Para adicionar a funcionalidade entregarTCC() ao sistema, basta colocar esse método na classe Estudante. O código fica assim:

```
class Estudante {
    String nome;
    public Estudante(String nome) {
        this.nome = nome;
    }
    public void estudar() {
        System.out.println(nome + " está estudando.");
    }
    public void entregarTCC(){
        //...
    }
}
```

Problemática

Você provavelmente já percebeu algo errado no código. Normalmente, estudantes de pós-graduação não entregam TCCs.

Só que a classe EstudanteDePosGraduacao é filha de Estudante, e portanto, deve apresentar todos os comportamentos dela.

Uma alternativa seria sobrescrever o método entregarTCC na classe EstudanteDePosGraduacao lançando uma exceção.

No entanto, continuaria sendo problemático: a classe EstudanteDePosGraduacao ainda não teria os comportamentos iguais aos de Estudante.

O ideal é que, nos lugares que estiver a classe Estudante, seja possível usar uma classe EstudanteDePosGraduacao, já que, pela herança, um estudante de pós-graduação é um estudante.

Solução

A solução para este problema é modificar a nossa modelagem. Podemos criar uma nova classe EstudanteDeGraduacao, que também herdará de Estudante. Essa classe terá o método entregarTCC:

```
public class EstudanteDeGraduacao extends Estudante {  
    public void estudar() {  
        System.out.println(nome + " está estudando na graduação.");  
    }  
    public void entregarTCC() {  
        //...  
    }  
}
```

Repare que, dessa forma, nossas classes representam melhor o mundo real. Não estamos forçando uma classe a fazer algo que ela originalmente não faz.

Além disso, se precisarmos utilizar uma instância de Estudante, podemos passar, sem medo, uma instância de EstudanteDeGraduacao ou de EstudanteDePosGraduacao.

Afinal de contas, essas classes conseguem executar todas as funções de Estudante — *mesmo tendo funções mais específicas*.

Definição do Princípio da Substituição de Liskov

Quem propôs o Princípio da Substituição de Liskov, de maneira formal e matemática, foi Bárbara Liskov.

No entanto, Robert Martin deu uma definição mais simples para ele: “*Classes derivadas (ou classes-filhas) devem ser capazes de substituir suas classes-base (ou classes-mães)*”.

Ou seja, uma classe-filha deve ser capaz de executar tudo que sua classe-mãe faz. Esse princípio se conecta com o polimorfismo e reforça esse pilar da POO.

É importante notar também que, ao entendermos esse princípio, passamos a nos atentar mais para o código: caso um método de uma classe-filha tenha um retorno muito diferente do da classe-mãe, ou lance uma exceção, por exemplo, já dá para perceber que algo está errado.

Se no seu programa você tem uma abstração que se parece com um pato, faz o som de um pato, nada como um pato, mas precisa de baterias, sua abstração está equivocada.

Imagine que, no seu projeto, você tem uma classe Pato, e uma classe filha dela, PatoBorracha. Nessa imagem você pode perceber que um pato e um pato de borracha são bem diferentes:



Se em uma parte do código você precisar usar um objeto Pato, mas usar um PatoBorracha no seu lugar, pode ter problemas.

Isso fere o princípio de substituição de Liskov, já que não conseguimos substituir um pai por um filho completamente.

Vantagens de aplicar o Princípio da Substituição de Liskov

Aplicar esse princípio nos traz diversos benefícios, especialmente para ter uma modelagem mais fiel à realidade, reduzir erros inesperados no programa e simplificar a manutenção do código.

➤ Princípio de Segregação de Interface (I - Interface Segregation Principle)

Para entender o Princípio de Segregação da Interface, imagine que estamos trabalhando com um sistema de gerenciamento de funcionários de uma empresa.

Vamos criar uma **interface**, conforme o código abaixo:

Interface Funcionário

```
public interface Funcionario {  
    public BigDecimal salario();  
    public BigDecimal gerarComissao();  
}
```

Repare que criamos a interface para estabelecer um “contrato” com as pessoas que são funcionárias dessa empresa. Nesse contexto, o código a seguir descreve duas classes que fazem referências a duas profissões nessa empresa: **Vendedor** e **Recepcionista**.

Ambas usam a interface **Funcionario** e, portanto, devem implementar os métodos `salario()` e `gerarComissao()`.

Classe Vendedor

```
import java.math.BigDecimal;  
public class Vendedor implements Funcionario {  
    @Override  
    public BigDecimal salario() {  
    }  
    @Override  
    public BigDecimal gerarComissao() {  
    }  
}
```

Classe Recepcionista

```
import java.math.BigDecimal;  
public class Recepcionista implements Funcionario {  
    @Override  
    public BigDecimal salario() {  
    }  
    @Override  
    public BigDecimal gerarComissao() {  
    }  
}
```

Problemática

Analisando o código acima, faz sentido uma pessoa que possui o cargo de vendedora ou recepcionista ter **salário**? **Sim!** Afinal, todos nós temos boletos para pagar.

Seguindo esta mesma linha, faz sentido uma pessoa com cargo de vendedor ou recepcionista ter **comissão**? **Não!**

Para uma pessoa que tem o cargo de vendedora, faz sentido. Mas para a pessoa que tem o cargo de recepcionista, não faz sentido.

Ou seja, a classe Recepcionista foi **forçada** a implementar um método que não faz sentido para ela. Embora ela seja funcionária dessa empresa, esse cargo não recebe comissão.

Portanto, podemos perceber que este problema foi gerado por termos uma **interface genérica**.

Solução

Para resolver isso, é possível criar **Interfaces específicas**. Ao invés de ter uma única interface Funcionário, podemos ter duas: Funcionario e Comissionavel.

Interface Funcionário

```
import java.math.BigDecimal;

public interface Funcionario {
    public BigDecimal salario();
}
```

Repare que mantemos a interface Funcionario, mas retiramos o método gerarComissao() a qual é específico de algumas pessoas, para adicioná-lo em uma nova interface FuncionarioComissionavel:

Interface Comissionável

```
import java.math.BigDecimal;

public interface Comissionavel{
    public BigDecimal gerarComissao();
}
```

Agora, a pessoa que possui o direito de ter comissão irá implementar a interface Comissionavel, um exemplo disso é a classe Vendedor:

Vendedor

```
import java.math.BigDecimal;

public class Vendedor implements Funcionario, Comissionavel{
    @Override
    public BigDecimal salario() {
    }
    @Override
    public BigDecimal gerarComissao() {
    }
}
```

Agora, a classe Recepcionista pode implementar a interface Funcionario sem ter a obrigação de criar o método gerarComissao():

Recepcionista

```
import java.math.BigDecimal;

public class Recepcionista implements Funcionario{
    @Override
    public BigDecimal salario() {
    }
}
```

Definição do Princípio da Segregação da Interface

Conforme analisamos o código acima, podemos perceber que:

Devemos criar **interfaces específicas** ao invés de termos uma **única interface genérica**.

E é justamente isto que **Princípio da Segregação da Interface** diz: *“Uma classe não deve ser forçada a implementar interfaces e métodos que não serão utilizados”*.

É possível que você já tenha comprado um adaptador com várias entradas, tal qual a foto. Na maioria das vezes, as pessoas não sabem a utilidade de todas as conexões.



Seguindo essa analogia, se não precisamos de um conector ou de uma entrada específica, não faz sentido incluí-los — assim como comprar um conector sob medida para um aparelho específico.

Ou seja, uma classe também não deve ser obrigada a implementar métodos que não serão utilizados.

Vantagens de aplicar o Princípio da Segregação da Interface

Seguir o Princípio da Segregação da Interface ajuda a promover a coesão e a flexibilidade em nossos sistemas, tornando-os fáceis de manter e estender.

➤ Princípio da Inversão de Dependência (D - Dependency Inversion Principle)

Para compreender o Princípio da Inversão de Dependência (letra O da sigla) imagine que estamos trabalhando em uma startup de e-commerce e precisamos desenvolver o sistema de gerenciamento de pedidos.

Sem conhecer o **Princípio da Inversão de Dependência**, é bem provável que vamos desenvolver uma classe `PedidoService` semelhante ao código abaixo:

Classe `PedidoService`

```
public class PedidoService {
    private PedidoRepository repository;
    public PedidoService() {
        this.repository = new PedidoRepository();
    }
    public void processarPedido(Pedido pedido) {
        // Lógica de processamento do pedido
        repository.salvarPedido(pedido);
    }
}
```

Problemática

Aparentemente, o código parece estar certo. No entanto, se um dia precisar alterar o armazenamento deste pedido para um outro lugar (por exemplo, uma API externa), vai precisar de mais de uma classe para resolver o problema.

Afinal, a **classe `PedidoService` está diretamente acoplada à implementação concreta da classe `PedidoRepository`.**

Solução

Para resolver este problema, podemos criar uma **interface** para a classe de acesso ao banco de dados e injetá-la na classe `PedidoService`.

Dessa forma, nós estamos **dependendo de abstrações e não de implementações concretas**.

Interface `PedidoRepository`

```
public interface PedidoRepository {
    void salvarPedido(Pedido pedido);
}
```

Classe `PedidoService`

```
public class PedidoService {
    private PedidoRepository repository;
    public PedidoService(PedidoRepository repository) {
        this.repository = repository;
    }
    public void processarPedido(Pedido pedido) {
        // Lógica de processamento do pedido
        repository.salvarPedido(pedido);
    }
}
```

Deste modo, conseguimos fazer com que a classe de **alto nível** (`PedidoService`) seja **independente** dos detalhes de implementação da classe de **baixo nível** (`PedidoRepository`).

Definição do Princípio da Inversão de Dependência

O **Princípio da Inversão de Dependência** diz: *“dependa de abstrações e não de implementações concretas”*.

Assim, é recomendado que os **módulos de alto nível** não dependam diretamente dos detalhes de implementação de **módulos de baixo nível**.

Em vez disso, eles devem depender de abstrações ou interfaces que definem contratos de funcionamento. Isso promove maior flexibilidade e facilita a manutenção do sistema.

Por exemplo, a funcionalidade de um equipamento eletrônico qualquer é garantida pela conexão adequada entre o plug e a tomada, não é mesmo?



Nessa analogia, os módulos de alto nível representam o plug, enquanto os módulos de baixo nível correspondem à tomada.

Da mesma forma que um plug se conecta à tomada independentemente de seus detalhes internos, os módulos de alto nível devem se vincular a abstrações ou interfaces, estabelecendo contratos de funcionamento.

Essa abordagem assemelha-se a usar um plug padronizado, garantindo uma conexão flexível e fácil manutenção.

Vantagens de aplicar o Princípio da Inversão de Dependência

A adesão ao Princípio de Inversão de Dependência promove a flexibilidade e a extensibilidade dos nossos sistemas.

Isso faz com que seja mais fácil fazer testes de unidade e construir códigos mais robustos e duradouros.

Agora que abordamos todos os conceitos do SOLID com exemplos práticos, vamos conhecer de que forma este princípio está ligado ao *Clean Code*.

Clean Code

No universo da programação, frequentemente nos deparamos com o termo: [Clean Code](#) ou Código Limpo.

Mas o que exatamente é um “código limpo”? Quais características são necessárias para obtê-lo?

Escrever um código limpo significa escrever códigos de um jeito que conseguimos entendê-lo sem complicação.

Isso não apenas simplifica a manipulação do código, mas também facilita a colaboração entre o time. No fim das contas, todo desenvolvimento e manutenção do sistema também se torna mais fácil.

De acordo com "Uncle Bob", em seu livro *“Código Limpo: Habilidades Práticas do Software Ágil”*, existem algumas boas práticas fundamentais para alcançar a clareza do código.

Vamos conhecê-las, a seguir:

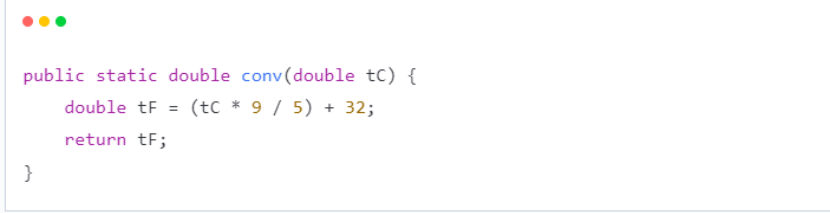
Utilizar os princípios SOLID:

O Clean Code e os princípios SOLID compartilham o objetivo de melhorar a qualidade do software, tornando-o legível, organizado, extensível e fácil de manter.

➤ Possuir nomes significativos

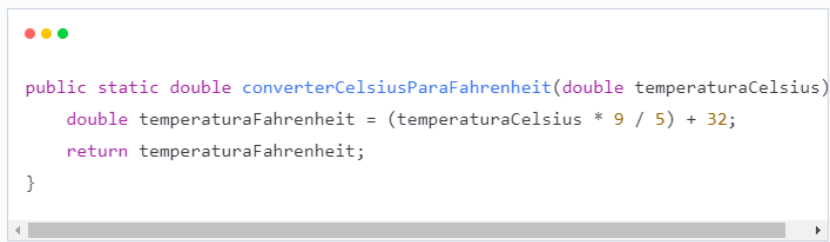
Nomes descritivos ajudam a entender a finalidade de uma parte do código sem a necessidade de comentários explicativos.

Para ilustrar, considere o código a seguir:



```
public static double conv(double tC) {  
    double tF = (tC * 9 / 5) + 32;  
    return tF;  
}
```

Temos que nos esforçar para entender o que o código acima faz. Podemos melhorar o entendimento apenas adicionando nomes significativos para as variáveis e para o método:



```
public static double converterCelsiusParaFahrenheit(double temperaturaCelsius)  
    double temperaturaFahrenheit = (temperaturaCelsius * 9 / 5) + 32;  
    return temperaturaFahrenheit;  
}
```

Agora, fica claro qual é o propósito do código, sem a necessidade de se lembrar de fórmulas ou realizar pesquisas adicionais. Isso economiza tempo e evita confusões desnecessárias.

➤ Priorizar o uso de funções pequenas

Escrever métodos ou funções pequenas e focadas em uma única tarefa é fundamental para manter o código claro e seguir o princípio da responsabilidade única (SRP).

Para ilustrar, considere o código a seguir:

```
public class Main {
    public static void main(String[] args) {
        int[] numeros = {1, 2, 3, 4, 5};

        int soma = 0;
        for (int numero : numeros) {
            soma += numero;
        }

        double media = (double) soma / numeros.length;

        if (media > 3) {
            System.out.println("A média é maior que 3");
        } else {
            System.out.println("A média é menor ou igual a 3");
        }
    }
}
```

Apesar do uso de nomes descritivos, a legibilidade poderia ser melhorada dividindo as tarefas em funções distintas, cada uma com sua descrição. Por exemplo:

```
public class Main {
    public static void main(String[] args) {
        int[] numeros = {1, 2, 3, 4, 5};

        int soma = calcularSoma(numeros);
        double media = calcularMedia(numeros);
        verificarEMostrarResultado(media);
    }

    public static int calcularSoma(int[] numeros) {
        int soma = 0;
        for (int numero : numeros) {
            soma += numero;
        }
        return soma;
    }

    public static double calcularMedia(int[] numeros) {
        return (double) calcularSoma(numeros) / numeros.length;
    }

    public static void verificarEMostrarResultado(double media) {
        if (media > 3) {
            System.out.println("A média é maior que 3");
        } else {
            System.out.println("A média é menor ou igual a 3");
        }
    }
}
```

Embora o código tenha ficado maior, ganhamos em legibilidade e segmentação. Qualquer pessoa que precise alterar a maneira como a média é exibida à pessoa usuária, só precisa modificar o método `verificarEMostrarResultado`.

Isso demonstra como funções pequenas podem facilitar a manutenção e a compreensão do código.

➤ Evitar comentários desnecessários

O código deve ser autoexplicativo, com nomes significativos e estrutura lógica clara. Comentários excessivos podem tornar o código poluído e difícil de manter.

Para ilustrar, considere o código a seguir:

```
public class R {  
    private double w;  
    private double h;  
    // Método para calcular a área  
    public double calc() {  
        return w * h;  
    }  
}
```

Os nomes curtos para as variáveis dificultam o entendimento, fazendo necessário o uso de comentários no nosso código, deixando o nosso código sujo. Então, podemos resolver isso adicionando nomes descritivos e removendo os comentários:

```
public class Retangulo {  
    private double largura;  
    private double altura;  
    public Retangulo(double largura, double altura) {  
        this.largura = largura;  
        this.altura = altura;  
    }  
    public double calcularArea() {  
        return largura * altura;  
    }  
}
```

Pronto, perceba como facilitou o entendimento. Qualquer pessoa desenvolvedora que ler este código consegue assimilar o que cada parte faz.

➤ Evitar complexidade

A complexidade desnecessária pode aumentar a chance de erros e tornar o código difícil de manter. Um exemplo de código complexo para fazer algo simples, como somar dois números, seria:

```
public void soma() {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Digite o primeiro número: ");
    String num1String = scanner.nextLine();
    System.out.print("Digite o segundo número: ");
    String num2String = scanner.nextLine();
    boolean validInput = false;
    double num1 = 0;
    double num2 = 0;
    while (!validInput) {
        num1 = Double.parseDouble(num1String);
        num2 = Double.parseDouble(num2String);
        validInput = true;
    }
    double soma = num1 + num2;
    System.out.println("A soma dos números é: " + soma);
    scanner.close();
}
```

Repare que é feita uma verificação da entrada, para só depois convertê-la em double.

Poderíamos simplesmente considerar que é esperado que o usuário digite um double e fazer o tratamento de exceções relacionado a isso:

```
import java.util.InputMismatchException;
import java.util.Scanner;
public class SimpleSumWithErrorHandling {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        try {
            System.out.print("Digite o primeiro número: ");
            double num1 = scanner.nextDouble();
            System.out.print("Digite o segundo número: ");
            double num2 = scanner.nextDouble();
            double soma = num1 + num2;
            System.out.println("A soma dos números é: " + soma);
        } catch (InputMismatchException e) {
            System.out.println("Erro: Por favor, digite números válidos.");
        } finally {
            scanner.close();
        }
    }
}
```

Executamos a lógica desejada de forma rápida e fácil de compreender. Imagine que precisamos somar agora 3 variáveis, ao invés de duas.

É mais fácil modificar o segundo código do que o primeiro. O segundo código é um exemplo de código limpo.

➤ Fazer o mínimo de argumentos

Funções e métodos devem ter o mínimo possível de argumentos. Isso melhora a legibilidade e a facilidade de uso.

No exemplo a seguir, note como cadastrar uma pessoa colaboradora é complexo ao passar muitos parâmetros:

```
cadastrarFuncionario(String nome, int idade, String cargo, double salario, Stri
de cadastro do funcionário aqui...
```

Ao chamar esse método, é difícil entender qual parâmetro utilizar em que lugar, podendo confundi-los, por exemplo.

Uma boa alternativa seria criar uma classe para representar o funcionário, outra para o endereço, e mais uma para o contato. Assim, faremos a divisão:

```
cadastrarFuncionario(Funcionario funcionario, Endereco endereco, Contato contat
```

Dessa forma, conseguimos agrupar as informações para que seja possível usar menos argumentos. Essa é uma boa prática

➤ Evitar código com repetição

A repetição torna o código difícil de manter, pois quando há mudanças necessárias elas precisam ser aplicadas em múltiplos lugares.

Então, extraia código repetido em funções ou métodos para promover a reutilização e a manutenção eficiente.

Um exemplo disso:

```
public static void main(String[] args) {  
    int numero1 = 5;  
    int numero2 = 7;  
    // Cálculo do fatorial para o primeiro número  
    int resultado1 = 1;  
    for (int i = 1; i <= numero1; i++) {  
        resultado1 *= i;  
    }  
    System.out.println("Fatorial de " + numero1 + ": " + resultado1);  
    // Cálculo do fatorial para o segundo número  
    int resultado2 = 1;  
    for (int i = 1; i <= numero2; i++) {  
        resultado2 *= i;  
    }  
    System.out.println("Fatorial de " + numero2 + ": " + resultado2);  
}
```

Como calculamos o fatorial mais de uma vez, extraímos o código para uma função, evitando repetições de cálculo no código:

```
public static void main(String[] args) {  
    int numero1 = 5;  
    int numero2 = 7;  
    // Cálculo e impressão do fatorial para o primeiro número  
    calcularEImprimirFatorial(numero1);  
    // Cálculo e impressão do fatorial para o segundo número  
    calcularEImprimirFatorial(numero2);  
}  
// Função para calcular e imprimir o fatorial de um número  
public static void calcularEImprimirFatorial(int numero) {  
    int resultado = 1;  
    for (int i = 1; i <= numero; i++) {  
        resultado *= i;  
    }  
    System.out.println("Fatorial de " + numero + ": " + resultado);  
}
```

Dessa forma, se precisarmos calcular novamente o fatorial de outro número, não precisaremos repetir código. Basta chamar a função de calcular fatorial novamente. Isso facilita o desenvolvimento.

➤ Vantagens de deixar o código limpo

Ao implementar cada um desses princípios e práticas, você não apenas irá melhorar a qualidade do seu código, mas também facilitará a compreensão, a manutenção e a colaboração no desenvolvimento de software. Tudo isso que discutimos aqui resume uma frase dita por Martin Fowler:

“Qualquer tolo pode escrever código que um computador pode entender. Bons programadores escrevem código que humanos podem entender”.

A programação não se trata apenas de fazer a máquina funcionar, mas também de criar soluções que sejam compreensíveis e colaborativas.

Conclusão

Neste artigo, conhecemos o significado da sigla SOLID e como cada um dos princípios ajudam a manter nossos projetos orientados a objetos mais coesos e limpos, atingindo o famoso *clean code*.

Agora você pode começar aplicar todos esses conhecimentos em seus projetos para aprimorar sua prática como dev!